Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 203 (2009) 55–69
www.elsevier.com/locate/entcs

DeSpec: Modeling the Windows Driver Environment [1]

Tomas Matousek[2]

Distributed Systems Research Group
Charles University in Prague
Prague, Czech Republic

Pavel Jezek[3]

Distributed Systems Research Group
Charles University in Prague
Prague, Czech Republic

Abstract

This paper introduces a new object-oriented specification and modeling language called DeSpec. The language targets primarily model checking in the Windows NT kernel driver environment. It integrates the majority of Zing modeling language features and adds means for defining parameterized abstractions of the environment at varying levels of detail. The DeSpec language also enables capturing constrains imposed on drivers by the Windows kernel in a form of quantified temporal logic patterns – easy-to-read templates of LTL formulae introduced by the Bandera toolset.

Keywords: specification language, verification, model checking, Windows kernel drivers

1 Introduction

In recent years, efforts to verify correctness of Windows kernel drivers [22] have emerged as it is crucial for stability of the whole operating system. Microsoft itself has developed several tools for driver verification including the latest Static Driver Verifier model checker. The key to successful application of the model checking approach in this area is a reasonable choice of the environment model. However, the environment models used in current tools are too (1) non-deterministic, degrading preciseness of the model checker reports, and (2) oversimplified, loosing the ability

[1] This work was partially supported by the Czech Academy of Sciences project 1ET400300504 and the Grant Agency of the Czech Republic project GD201/05/H014.

[2] Email: matousek@nenya.ms.mff.cuni.cz

[3] Email: jezek@nenya.ms.mff.cuni.cz

doi:10.1016/j.entcs.2009.03.026

to check more specific kinds of properties of drivers. On the other hand, neither a formal or readable specification usable for documentation purposes is provided by these models. This paper targets these issues by introducing a new language for formal specification and modeling of kernel drivers and their environment.

Please note that due to space limitations the paper presents only a small excerpt of the language features. The full language specification, detailed elaboration of its features and also a large sample specification of the Windows environment can be found in [15].

1.1 Model Checking

Model checking technique is a formal verification method based on thorough examination of a program model state space. The model reflects behavior of the program related to the property being verified. It should ideally retain any part of the software that might influence the property so that the verification is sound and complete. On the other hand, the model should be as simple as possible since the model checker has to explore all its possible states. The time and space requirements for the verification are growing exponentially with respect to the number of operations, threads and variables used in the model (the *state explosion problem* [17]).

Usually, the goal is not to model check the system as a whole. Instead, the system is split into two pieces – a particular component of interest (a module, also an open system [32]) and the rest of the system (the module's environment). The environment is considered correct and its provided and required interfaces are defined by a specification. The verification tool is expected to extract a partial model from the module's source code and complete it by including the environment's behavior model according to the specification. The resulting model passed to the model checker, captures the module's interaction with the environment relevant to a set of properties being verified.

The process of model extraction from the program's source code is a difficult task as the source code language itself can induce major problems. A C language extractor needs to understand constructs like pointers, arrays, unions, reinterpreting type casts, etc. Fortunately, even though some of these constructs in general allow the extractor to build neither sound nor complete model of the program, results of the software verification are still valuable. All the issues of the model extraction from the C source code have already been presented in a different paper [16].

This paper focuses on a formal description of the environment that combines the requirements on the module and modeling of the functionality provided by the environment. Temporal logics (e. g. *Linear Temporal Logic* (LTL) [14]) are often used for the former. They define how properties of the system should change in time using predicates quantified over time variable. However, specifying properties of a real application by means of plain temporal logic comes with a significant drawback. The specification is not easy to comprehend for the most of driver programmers and if a formula gets more complex neither for temporal logic experts.

Plain logic unreadability drives efforts to develop a higher-level language like

Bandera *Temporal Logic Patterns* [9]. Properties expressed in this language are translated into the temporal logic formulae consumed by many existing verification tools. The patterns allow writing frequently used temporal logic formulae in very simple plain English sentences, e. g. "P is absent between Q and R" is representing the $\Box((Q \wedge \neg R \wedge \Diamond R) \Rightarrow [P\ U\ R])$ formula. Though incomplete the patterns are sufficient to specify widely used properties. Moreover, additional patterns can be added to the repertoire if needed.

In our work, we use *Zing* modeling language [1,24] as a basis for specification of the environment behavior and also as the output language of the model extractor. Zing language and Zing model checker have been developed by Microsoft Research group. The choice was made due to Zing's rich modeling functionality and the state of its current development (the preview implementation is available and works quite well). However, most ideas behind this work are not dependent on the target modeling language and can be applied to any other modeling language that provides at least classes, methods, exceptions, non-deterministic choices, and threads. Another modeling language meeting these criteria should be the new version of *Bandera Intermediate Representation* (BIR) – a modeling language of *Bogor* model checking framework [28].

1.2 Checking Windows Drivers

Windows kernel drivers are relatively small libraries usually written in the C language. They run in a privileged mode that enables them to work directly with hardware. This introduces a high risk of damaging other parts of the kernel if a driver contains an error. Hence the correctness of drivers is crucial for security and stability of an operating system and drivers are common subject of software verification.

A driver can be seen as a component put into the environment comprising of the kernel and other drivers. Since drivers usually communicate with each other only via kernel function calls the inclusion of the other drivers into the environment is an acceptable simplification. The verifier deals with the open system verification as the source code of the Windows kernel is usually not available. And even if it was, it would be virtually impossible to extract and verify the kernel model due to its inherent complexity. Besides, drivers shouldn't depend on the exact behavior of the private parts of the kernel as they can change version to version. Only the public documented functionality should be relied on.

So the model extractor should only work with the kernel specification. However, such specification is not currently available in a form that would be feasible to drive the model extractor – the only source of official documentation is the *Driver Development Kit* (DDK) [21] provided by Microsoft, where the rules the drivers should comply with are described in plain English and some important details are stated vaguely or even missing entirely. It is a goal of our work to provide a language for writing the specification and to apply it on significant parts of the kernel API.

Several tools that verify driver correctness have already been developed by Microsoft itself. These include the *Driver Verifier* [18] tool for run-time driver verifica-

tion, the *PREfast* [19] static analysis tool based on local analysis of driver functions and finally the *Static Driver Verifier* (SDV) [20] (still in development) based on techniques of static analysis of the whole driver and model checking.

The Static Driver Verifier (SDV) models the kernel environment in C language enriched with special functions and macros that handle non-determinism necessary for emulating various execution paths. The rules the drivers can be verified against are written in *Specification Language for Interface Checking* (SLIC) [2]. Expressing a rule in the SLIC language inheres in writing pieces of C pseudo-code and defining how the environment model should be instrumented by them. The resulting instrumented code is converted to an abstract Boolean program which is passed to the model checker. The very first Boolean program extracted from the instrumented code abstracts from all local variables and replaces all conditions by non-deterministic choices. Error traces are then discovered by the model checker and confronted with the original program via symbolic execution. If an error trace describes the execution that is actually infeasible, the Boolean program is refined to be more specific with respect to the variables influencing the trace. The refined program is passed back to the model checker. This process of error search and model specialization repeats until there are no infeasible error traces found or a timeout elapses.

The environment model and the SLIC language allows safety properties to be checked with respect to operations performed sequentially on a single device object (an object representing a device in the driver). SLIC rules are limited to safety properties so it is not possible to encode all the rules defined in the DDK. The rules are specified separately from kernel environment which makes them less maintainable. Inability to model multi-threaded environment and simultaneous work on more device objects also prevents from verification of some race conditions commonly contained in faulty Windows drivers. In this work we introduce a solution that does not have these shortcomings.

1.3 Paper Contribution

The aim of this work is to make it possible to specify and model the kernel environment in a formal yet comprehensible form, which could be used not only for precise documentation of the kernel API but above all as an input for a model extractor that produces verifiable concurrent models of the Windows drivers. For this purpose, the paper introduces a new specification and modeling language called *Driver Environment Specification Language* (abbreviated as *DeSpec*). As shown in [15], the language is able to capture a significant subset of the rules imposed on drivers by the DDK including those that are difficult or impossible to express in the SLIC language and hence currently not verifiable by the SDV.

The rest of the paper is laid out as follows. Section 2 briefly describes the Windows kernel environment from a point of view of the driver verification. Section 3 introduces the DeSpec language, explains its part on an example and describes how a model extractor should work with DeSpec specifications. Section 4 discusses related work and Section 5 concludes.

2 Windows Kernel Environment

The Windows kernel executive comprises of several components that manage various system resources – the managers [31]. The managers provide services for the other parts of the executive and for drivers. The *I/O Manager*, the *Plug & Play Manager*, and the *Power Manager* are the ones that are most interesting for driver verification as they do the majority of communication with drivers. Note, this work is limited only to drivers following the *Windows Driver Model* (WDM) [26]. Such drivers have to implement Plug & Play and power management features.

The I/O Manager loads and unloads drivers and issues I/O requests on them. The drivers are directly controlled by the I/O Manager, which issues I/O requests in form of *I/O Request Packets* (IRPs). If a driver can complete the request it fills in a place in the packet reserved for output parameters and passes the packet back to the manager. If it doesn't implement the required functionality it can pass the request to an optional lower level driver – a hierarchy is being formed by such inter-driver relationships. The other managers issue their requests and notifications to the drivers through the I/O Manager. For example, the Plug & Play Manager keeps track of the device state transitions (device removal, stopping, starting, etc.) and the Power Manager monitors the power state of the machine (whether it is going to sleep, awaking, etc.). Both managers notify the driver appropriately by sending it the respective IRPs.

Each driver has to respond correctly to an arbitrary request and content of the packet. It can return a result indicating an error, but it must never crash or damage other parts of the kernel. The driver cannot make any assumptions about drivers above or below it in the hierarchy. This requirement allows the verification tool to isolate the driver and test it on arbitrary inputs and outputs from the I/O Manager and higher/lower level drivers.

3 Driver Environment Specification Language

3.1 Overview

The Driver Environment Specification Language (DeSpec) is an object-oriented specification and modeling language incorporating the majority of features of the Zing modeling language [1] combined with design-by-contract elements inspired by Spec# language [3], and Bandera Temporal Logic Patterns [9]. It is designed to guide extraction of Zing models from source code of Windows kernel drivers.

DeSpec language allows modeling of I/O Manager's behavior to drivers, modeling of kernel functions behavior and specifying constrains and rules that drivers should obey when calling these functions. Models and abstractions can be defined in various levels of detail, which, as one of the solutions fighting against the state space explosion problem, enables the model extractor to infer the smallest available model sufficient for the verification of a particular rule.

DeSpec language provides means for capturing basic elements of the interaction between driver and its environment (i. e. global variables, functions and data struc-

tures). It is designed as a bridge between constructs of the C language and their models in the Zing language. In particular the models of pointers, function pointers, unions and other constructs that are not directly expressible in the Zing language are hidden behind the syntax of DeSpec language. This allows to adjust models for these features without a need to rewrite the specifications.

Apparently, some constructs exploiting memory layout, such as reinterpreting casts or unions, cannot be modeled in a feasible way. Therefore they are not directly expressible in the DeSpec language. Fortunately, the driver as well as environment interface should be as platform independent as possible and thus these constructs should be used rarely.

3.2 Structure of Specifications

The DeSpec language is similar to the C# language in its syntactical structure. Each source file contains a list of declarations grouped to namespaces. Declarations include classes, integer enumerations, integer ranges, method delegates and method groups. A class declaration comprises of its members. Apart from fields and methods, which are common for standard object-oriented languages, DeSpec classes can also contain rules. A rule specifies constrains on fields and methods by means of temporal logic patterns. This section briefly describes DeSpec namespaces, classes and rules.

3.2.1 Namespaces

A namespace defines a scope for abstractions of kernel functions and structures. When the model extractor searches for an abstraction of a kernel function or a structure used in the driver's source code it looks up a single namespace only. The choice of the namespace depends on the constrains to be verified. The default (global) namespace describes a minimal model for kernel functions and structures. Other namespaces usually *refine* the default model – making it more complex to enable verification of a constraint not expressible by means of default model. Constraints are embedded into the specification as method preconditions, postconditions, type constrains, rules, etc. By choosing the constraint to verify, the containing namespace is designated for being searched by the extractor. The ability to differentiate specifications by level of details is important for reducing the size of the resulting model.

3.2.2 Classes

Although Windows kernel is written in the C programming language its design is object oriented. Usually, a structure representing an object within the kernel (e. g. a semaphore, mutex or device) is provided along with functions working with it. These functions behave like methods of the structure (object) as they all take a pointer to the structure as one of their parameters (the "this" reference). A notion of inheritance is also present on several places. Inheritance is used for sharing data among structures representing different yet related objects. The sharing technically inheres in declaring common initial fields in the related structures.

These observations justify introduction of classes as main elements of the specifications – the kernel structures provided to drivers are modeled in DeSpec as classes. The functions bound to these structures are declared as class instance methods. Functions not bound to any instance are mapped to static methods. The formal parameter referring to the instance the method is working on is specified by the *instance* keyword. The method (whether static or instance) abstracting a kernel function has to have the same name as the kernel function and no other method in the same namespace can have the same name (even though declared in another class). This rule allows the model extractor to find a specification of a function whose call has been observed in the source code. An example of a class specification follows:

Example 1.

```
class DEVICE_OBJECT {
  NTSTATUS IoAttachDevice(instance,_,out DEVICE_OBJECT attachedTo)
    requires !Driver.IsLowest;
  {
    NTSTATUS status = choose
    {
      NTSTATUS.STATUS_SUCCESS,
      NTSTATUS.STATUS_INVALID_PARAMETER,
      NTSTATUS.STATUS_OBJECT_TYPE_MISMATCH,
      NTSTATUS.STATUS_OBJECT_NAME_INVALID,
      NTSTATUS.STATUS_INSUFFICIENT_RESOURCES
    };
    attachedTo = IsSuccessful(status) ? Driver.LowerDevice : null;
    return status;
  }

  DEVICE_OBJECT IoAttachDeviceToDeviceStack(instance,_)
    requires !Driver.IsLowest;
  {
    return (choose(bool)) ? Driver.LowerDevice : null;
  }

  void IoDetachDevice(instance);

  /* more members follow */
}
```

In Example 1, the `DEVICE_OBJECT` class abstracts the structure of the same name. Instances of the structure represent devices that drivers are working with. Both *IoAttachDevice* and *IoAttachDeviceToDeviceStack* kernel functions attach the device object to the top of the device objects chain. The immediate lower device

object, where the instance is attached to, is returned in the *attachedTo* output argument and in the return value, respectively. The *IoDetachDevice* simply detaches the immediate higher level device from this device object instance [4] .

The signature of a method abstracting a kernel function defines how parameters of the function are treated within the specification. The *placeholder* token (a single underscore) is used for arguments that are not important for the specification. The models of *IoAttach-* functions do not care about the second parameter. When a specification refers to the *IoAttachDevice* method, only one argument is stated in the list of actual arguments. The instance argument is picked from the argument list out before the method to denote the target instance using the dot notation. Arguments on the positions of placeholders are also omitted in the actual argument list. Methods declared in Example 1 are referred to as follows:

```
device.IoAttachDevice(out lower_device)
device.IoAttachDeviceToDeviceStack()
lower_device.IoDetachDevice()
```

The *out* keyword specifies that the argument is an output argument and has to be assigned within the method's body. The output argument is mapped to the C language by an additional level of indirection. The C type of the argument is thus `DEVICE_OBJECT**`. The *ref* keyword is also supported for marking arguments passed in and out by reference.

A possibly empty list of preconditions and postconditions follows the signature. The syntax is similar to the one used in the Spec# language – the conditions are introduced by *requires* and *ensures* keywords, respectively. The condition is a Boolean expression with some limitations on the terms. The conditions stated in Example 1 require the lowest level driver not to call the *IoAttach-* functions. Pre- and postconditions are translated to assertions when the Zing model of the method is generated.

The body defines a model of the method's behavior using Zing syntax enriched with additional constructs that are translated to the Zing when the resulting model is generated. In Example 1, extended forms of the Zing's *choose* operator are used. Type `NTSTATUS` is an integer enumeration abstracting the kernel type of the same name. The operator *IsSuccessful* determines whether a value is a successful value of its type as recognized by the kernel.

The body can also be omitted at all if the modeled function does nothing that influences the driver at the current level of abstraction and only its calls are significant. If a kernel function returns some value to the caller (via a return value or output parameters), throws an exception or has some side-effect the specification method should have a body that models these operations.

Since a DeSpec class is usually an abstraction of a public kernel structure, it may contain fields corresponding to the fields of the structure. Additional fields that do not correspond to real fields might be necessary for storing auxiliary data

[4] Note the reverse roles of the device objects – the higher level device object is attaching but the lower level device object is detaching.

used only for the sole purpose of modeling. Such fields are marked by the *synthetic* keyword. Similarly, *synthetic methods* and also *synthetic classes* can be defined in the specification. In general, DeSpec distinguishes synthetic language elements from non-synthetic ones. Note that all elements used in the first example are non-synthetic. Synthetic classes contain no abstractions, particularly no kernel function is mapped to a method of a synthetic class. Example of a class containing synthetic attributes follows:

Example 2.

```
static class Driver {
  synthetic DEVICE_OBJECT LowerDevice = new DEVICE_OBJECT;

  [ModelParam]
  synthetic const bool IsLowest = false;

  /* more members follow */
}
```

In Example 2, two synthetic fields are defined in the static class. The first one, *LowerDevice*, is used as a dummy device object that all devices of the current driver are attached to. The model can abstract from the precise device objects chain because the drivers shouldn't care about what drivers are layered beneath them in the chain. Similar simplifications are necessary to reduce the size of the generated model.

The second field named *IsLowest* is a literal constant field defining whether or not the driver is the lowest level driver in the driver chain. The field is annotated by the *ModelParam* attribute, which means that its initial value should be set by the user prior to the model extraction. Model parameterization is utilized when the model depends on a property that is difficult to deduce automatically from the driver's source code. It can be also used for model size tuning.

3.2.3 Rules

Another member that can be present in the class is a *rule*. The rule is a list of quantified temporal logic patterns [9] with pattern parameters filled with Boolean expressions.

Example 3.

```
class DEVICE_OBJECT {
  /* method declarations from Example 1 omitted */

  static rule
    forall(DEVICE_OBJECT device)
    {
       _.IoAttachDevice(out device)::succeeded ||
```

```
      (device === _.IoAttachDeviceToDeviceStack()) && device!=null
    }
    corresponds to
    {
      device.IoDetachDevice()
    }
    globally;
}
```

The rule in Example 3 is a single pattern, however, in general, a rule is a list of quantified temporal logic patterns separated by commas and ending by a semicolon. The rule presented has the following meaning: "Each successfully attached device is eventually detached and each device that is detached has previously been successfully attached." Rest of the section explains the patterns in more detail.

Each temporal logic pattern is formed by pattern keywords and pattern expressions. The pattern used in Example 3 can be generalized to $\{P\}$ *corresponds to* $\{Q\}$ *globally*, where P and Q are Boolean expressions. Each pattern can be split into two parts: the property and the scope. In this case, the property is $\{P\}$ *corresponds to* $\{Q\}$ and the scope is *globally*. A list of available pattern properties follows:

(i) $\{Q\}$ *is universal*
(ii) $\{Q\}$ *is absent*
(iii) $\{Q\}$ *exists*
(iv) $\{Q\}$ *precedes* $\{R\}$
(v) $\{Q\}$ *leads to* $\{R\}$
(vi) $\{R\}$ *responds to* $\{Q\}$
(vii) $\{Q\}$ *corresponds to* $\{R\}$

Properties (i) to (vi) are defined in [9]. Properties (v) and (vi) are equivalent and it depends on the situation which one is more appropriate to use. The property (vii) is equivalent to a conjunction of properties (v) and (iv), i. e. to $\{Q\}$ *leads to* $\{R\}$ $\wedge$ $\{Q\}$ *precedes* $\{R\}$. It has been introduced to the language since the combination is frequently used in the kernel environment and it would be inconvenient to write the two patterns separately. The available scopes are:

(i) *globally*
(ii) *before* $\{S\}$
(iii) *after* $\{P\}$
(iv) *after* $\{P\}$ *until* $\{S\}$
(v) *between* $\{P\}$ *and* $\{S\}$

The meaning of each property and scope is obvious. Detailed definitions can be found in [9] along with the equivalent LTL formulae. The LTL formulae for

Method event operator	**Effect**
`M(args)::entered`	Returns *true* after *M* is entered with specified arguments until *M* returns. Return value is ignored.
`M(args)` `M(args)::returned`	Returns *true* after *M* is called with specified arguments and with any return value until *M* is entered again with the same arguments.
`M(args)::succeeded`	Returns *true* after *M* is called with specified arguments and with a successful return value until *M* is entered again with the same arguments.
`M(args)::failed`	Returns *true* after *M* is called with specified arguments and with an unsuccessful return value until *M* is entered again with the same arguments.
`expr === M(args)` `expr !== M(args)`	Returns *true* after *M* is called with specified arguments and with a return value (un)equal to the value of the *expr* until *M* is entered again with the same arguments.

Table 1
Source code event operators for methods. *M* stands for non-synthetic methods, *args* stands for a list of arguments (possibly empty), and *expr* denotes an expression.

Q-corresponds-to-R pattern with the global scope is

$$\Box(Q \Rightarrow \Diamond R) \wedge [R \ W \ Q]\,^{5}\,.$$

Temporal patterns can be quantified over value types or reference types. Patterns of instance rules are implicitly quantified by a variable of the declaring type. Instance rules can refer to that variable by using *this* keyword. This keyword can be omitted when referring to the instance members of the type. Unlike Bandera [8], DeSpec allows to quantify over value types (i. e. integers, Boolean, enumerations). Zing symbolic value types can be used for the implementation. The reference type quantification may be implemented in the same way as in the Bandera, however more scalable implementation would be possible using Zing symbolic reference types, which should be available in the next version of the Zing.

Boolean expressions comprising pattern parameters should refer to so called *source code events* via *source code event operators*. A source code event refers to an execution of a particular piece of code. DeSpec allows to specify events corresponding to function calls and operations on fields (read and write) within the

[5] $\Box$ is the universal time quantifier (always in the future), $\Diamond$ is the existential time quantifier (sometime in the future), $[\varphi \ W \ \psi]$ is the weak until operator (either ψ never holds and φ holds always, or ψ holds sometime in the future and φ holds until that moment).

Field event operator	**Effect**
`F::read`	Returns *true* after *F* is read from.
`F::written`	Returns *true* after *F* is written to.
`expr === F::get` `expr !== F::get`	Returns *true* after *F* is read from and the value read is (not) equal to the value of the *expr*.
`expr === F::set` `expr !== F::set`	Returns *true* after *F* is written to and the value written is (not) equal to the value of the *expr*.

Table 2
Source code event operators for fields. *F* stands for a non-synthetic field and *expr* denotes an expression.

driver's source code. Hence, source code event operators are applicable on non-synthetic methods and fields only. Available source code event operators are listed in Table 1 and Table 2.

In Example 3, the source code event defined by the method(args)::*succeeded* operator establishes a watchdog for successful returns from the kernel function *IoAttachDevice*. It is triggered by only such function return that the third argument can be unified with the *device* quantification variable and the function return value means a successful call. The first two arguments could have been arbitrary when the function was called.

Each source code event operator is replaced by the corresponding predicate for the purpose of rule verification. The use of the source code event operator inside a pattern expression implies adding a global state variable to the resulting Zing model and instrumentation of the model with pieces of Zing code that make transitions of the state. The value of the operator state variable determines the value of the LTL formula predicate. Although Zing doesn't support LTL verification directly, it is possible to use run-time verification algorithm proposed by [11].

3.3 DeSpec Driven Model Extraction

Inputs to the model extraction process are the source code of the driver being verified, kernel header files, and the specifications of kernel functions and data structures written in DeSpec. At the beginning, the user should select a set of constrains that he or she wants to verify.

The user also chooses the *top-level model* to be used for the verification. This model is also written in DeSpec as a class implementing the predefined methods. Its task is to emulate the kernel's behavior to the driver including driver loading and initialization and issuing I/O requests (IRPs). Default top-level model is the most complex one. It emulates multiprocessor environment, multiple device objects, and concurrent IRPs. However, for a verification of some rules a simpler model may be sufficient. DeSpec allows to write and use such model. The choice of the

simpler model may radically reduce the size of state space and make the verification faster and sometimes even allow the verification to be completed in realistic time. However, some errors may remain undiscovered.

Once the top-level model is chosen, the model extractor generates Zing model of the driver (using its C source code and kernel headers) and combines it with the environment model. Since the resulting model is too large to be verified, the slicing [13,10] should take place retaining only those parts transitively referred to by the top-level model and the constrains being verified. As a final result, a Zing model of the driver and the related kernel functions and structures are output.

4 Related Work

This work incorporates or relies on ideas and approaches of model checking [6,17], model extraction [7,30], temporal logics [27,14,5], source code static analysis and slicing [13,10], and Windows kernel driver environment [31,26].

In particular, the Zing model checker [1], Bandera toolset (especially the Bogor model checking framework [28,29]), *Java Path Finder* [25], and *SPIN model checker* [12,4] are related tools devoted to the model checking.

The *SLAM project* [23] is addressing the static analysis and verification of the C programs, especially the Windows kernel drivers. The beta version of Microsoft Static Driver Verifier (SDV) tool [20] has been recently released as a result of efforts in this area. Since this paper targets on Windows kernel drivers verification, the SDV is the closest related work. The way how rules are specified in this tool limits its verification power to safety properties. The environment model used by SDV is single-threaded, preventing verification of some race conditions, and quite non-deterministic, introducing additional false reports. It neither provides a specification of the kernel functions that might be used as a documentation. On the other hand, SDV is a functional tool whose application in practice already led to discovering several errors in Microsoft's own drivers.

Finding errors in drivers is not limited to the model checking technique. Microsoft *PREfast* tool for drivers [19] performs static analysis of the source code and searches for common error patterns. It can, for example, find memory leaks incurred by missing function calls, dereferences of null pointers, buffer overruns, kernel functions called on incorrect IRQL level, and so on. The analysis is function scoped and hence it introduces false negatives and also restricts a set of errors it is able to detect.

The Windows operating system also enables to check how drivers work in stress conditions such as lack of memory, missing resources, lost packets, etc. In cooperation with the kernel, *Driver Verifier* tool [18] emulates such conditions and runs tests on the specified driver. The tool is able to detect many errors but it doesn't do any static verification so many execution paths remain unchecked.

5 Conclusion and Future Work

This paper introduces the DeSpec language – a new specification and modeling language designed to enable writing modular, readable, and well arranged specifications of the Windows kernel driver environment as well as formally, yet still comprehensibly, capture rules imposed on drivers by the kernel and documented in plain English in DDK.

Expressiveness and suitability of the language are demonstrated on a part of the kernel functionality in [15]. This work also shows that the available documentation of the kernel environment [21] is not sufficient for its formal specification without a deeper understanding of the Windows kernel.

As the DeSpec language is intended to be utilized by model checking tools, it addresses the main issue of this verification method – the state explosion problem. The abstractions may vary in the level of detail chosen according to the properties being verified. Complexity of the model can be further tuned by the user specified model parameters. By setting these parameters, the user can influence how complex the extracted model will be and what may it neglect. The user may also select a subset of tested driver functionality by choosing an appropriate top-level model.

The possibility of verifying LTL formulae with finite trace semantics using assertions only (see [11]) arises a question whether the use of temporal rules brings something new beyond the use of explicit assertions. Although many rules may be equivalently verified manually, i. e. by adding assertions (or method contracts) on the right places in the functions' model code, the use of rules has some advantages. Several advantages are implied by the locality. If entire "business logic" of the rule is written on a single place it is easier maintainable, more readable, and the verification of the rule can be easier (un)selected for verification. Besides, when the rule is more complex it wouldn't be easy to manually keep track of all operations in the code that influences the verified property. On the other hand, some rules are too complicated to write or comprehend that it is better to implement them manually by explicit assertions.

The ideas proposed by this paper are currently being implemented. The implementation comprises of the DeSpec language analyzer and a model extractor consuming C source code and producing a Zing model driven by DeSpec specifications.

References

[1] Andrews, T., Qadeer, S., Rajamani, S. K., Rehof, J., Xie, Y: Zing: A model checker for concurrent software, Technical report, Microsoft Research, 2004.

[2] Ball T., Rajamani, S. K.: SLIC: a Specification Language for Interface Checking, Technical Report, MSR-TR-2001-21, Microsoft Research, 2002

[3] Barnett, M., Leino, K. R. M., Schulte, W.: The Spec# Programming System - An Overview, Microsoft Research, 2004

[4] Bell Labs: SPIN model checker, http://spinroot.com

[5] Clarke, E. M., Emerson, E. A., Sistla, A. P.: Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Transactions on Programming Languages & Systems, 244-263, 1986

[6] Clarke, E. M., Grumberg, O., Peled, D. A.: Model Checking, MIT Press, 2000.

[7] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, Zheng, H.: Bandera: Extracting Finite-state Models from Java Source Code, proceedings of the International Conference on Software Engineering (ICSE), 2000

[8] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Robby: Expressing Checkable Properties of Dynamic Systems: The Bandera Specification Language, 2001

[9] Dwyer, M. B., Avrunin, G. S., Corbett, J. C.: Patterns in property specifications for finite-state verification, in Proceedings of the 21st international Conference on Software Engineering, 411-420, 1999

[10] Dwyer, M. B., Hatcliff J.: Slicing Software for Model Construction, Journal of High-order and Symbolic Computation, 2000

[11] Giannakopoulou D., Havelund K.: Runtime Analysis of Linear Temporal Logic Specifications, RIACS Technical Report 01.21, 2001

[12] Holzmann, G. J.: The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley Professional, 2003

[13] Krinke, J.: Advanced Slicing of Sequential and Concurrent Programs, PhD thesis, Fakultt Fr Mathematik und Informatik, Universitt Passau, 2003

[14] Lamport, L.: "Sometime" is sometimes "not never" – on the temporal logic of programs, in Proceedings of 7th ACM Symposium on Principles of Programming Languages, pages 174-185, 1980.

[15] Matousek, T.: Model of the Windows Driver Environment, Master Thesis at Department of Software Engineering, Charles University in Prague, 2005, http://nenya.ms.mff.cuni.cz/publications/Matousek-thesis.pdf

[16] Matousek, T., Zavoral, F.: Extracting Zing Models from C Source Code, SOFSEM 2007

[17] McMillan, K. L.: Symbolic model checking – an approach to the state explosion problem, PhD thesis, SCS, Carnegie Mellon University, 1992

[18] Microsoft: Driver Verifier, http://www.microsoft.com/whdc/DevTools/tools/DrvVerifier.mspx

[19] Microsoft: PREfast, http://www.microsoft.com/whdc/devtools/tools/PREfast.mspx

[20] Microsoft: Static Driver Verifier – Finding Driver Bugs at Compile-Time, WHDC, http://www.microsoft.com/whdc/devtools/tools/sdv.mspx

[21] Microsoft: Windows Driver Development Kit, WHDC, http://www.microsoft.com/whdc/devtools/ddk/default.mspx

[22] Microsoft: Windows Driver Foundation, WHDC, http://www.microsoft.com/whdc/driver/wdf/default.mspx

[23] Microsoft Research: SLAM Project, http://research.microsoft.com/slam

[24] Microsoft Research: Zing Model Checker, http://research.microsoft.com/zing

[25] NASA Intelligent Systems Division: Java Path Finder, http://ase.arc.nasa.gov/havelund/jpf.html

[26] Oney, W.: Programming the Microsoft Windows Driver Model, 1999, Microsoft Press

[27] Pnueli, A.: The temporal logic of programs, in 18th IEEE Symposium on Foundation of Computer Science, pages 46-57, 1977.

[28] Robby, Dwyer, M. B., Hatcliff, J.: Bogor: An Extensible and Highly Modular Software Model Checking Framework, SIGSOFT Softw. Eng. Notes 28, 5, 267-276, 2003

[29] Robby, Dwyer, M. B., Hatcliff, J.: Bogor, http://bogor.projects.cis.ksu.edu

[30] SAnToS laboratory: Bandera project, http://bandera.projects.cis.ksu.edu

[31] Solomon, D. A., Russinovich, M. E.: Inside Microsoft Windows 2000 Third Edition, Microsoft Press, 2000

[32] Vardi, M. Y., Verification of Open Systems, Lecture Notes in Computer Science, Volume 1346/1997, Springer.